AFRL-RY-WP-TR-2012-0137

# BLACKJACK

Jack Dongarra, Antonios Danalis, and Piotr Luszczek

University of Tennessee Knoxville

Jeffrey Vetter and Gabriel Marin

Oak Ridge National Laboratory

Eric Stoltz and Michael Wolfe

The Portland Group

MAY 2012
Final Report

AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

# NOTICE AND SIGNATURE PAGE

*//signature//

ALFRED J. SCARPELLI, Project Engineer
Integrated Circuits & Microsystems Branch
Aerospace Components Division

//signature//

BRADLEY J. PAUL, Chief
Integrated Circuits & Microsystems Branch
Aerospace Components Division

//signature//

BRADLEY CHRISTIANSEN, Lt Col, USAF
Deputy Division Chief
Aerospace Components Division
Sensors Directorate

| REPORT DOCUMENTATION PAGE | *Form Approved OMB No. 0704-0188* |
|---|---|

| | | |
|---|---|---|
| **1. REPORT DATE** *(DD-MM-YY)* May 2012 | **2. REPORT TYPE** Final | **3. DATES COVERED** *(From - To)* 09 March 2009 – 02 April 2012 |

| | |
|---|---|
| **4. TITLE AND SUBTITLE** BLACKJACK | **5a. CONTRACT NUMBER** FA8650-09-C-7916 |
| | **5b. GRANT NUMBER** |
| | **5c. PROGRAM ELEMENT NUMBER** 62303E |
| **6. AUTHOR(S)** Jack Dongarra, Antonios Danalis, and Piotr Luszczek (University of Tennessee Knoxville)<br>Jeffrey Vetter and Gabriel Marin (Oak Ridge National Laboratory)<br>Eric Stoltz and Michael Wolfe (The Portland Group) | **5d. PROJECT NUMBER** 3000 |
| | **5e. TASK NUMBER** YD |
| | **5f. WORK UNIT NUMBER** Y0H4 |
| **7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**<br>University of Tennessee Knoxville<br>800 Andy Holt Tower<br>Knoxville, TN 37996-0003<br>Oak Ridge National Laboratory<br>------------------------------<br>The Portland Group | **8. PERFORMING ORGANIZATION REPORT NUMBER** |
| **9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)**<br>Air Force Research Laboratory<br>Sensors Directorate<br>Wright-Patterson Air Force Base, OH 45433-7320<br>Air Force Materiel Command<br>United States Air Force | **10. SPONSORING/MONITORING AGENCY ACRONYM(S)** AFRL/RYAP |
| | **11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)** AFRL-RY-WP-TR-2012-0137 |

**12. DISTRIBUTION/AVAILABILITY STATEMENT**
Approved for public release; distribution unlimited.

**13. SUPPLEMENTARY NOTES**
PAO Case Number: DARPA-19767; cleared 21 August 2012. Report contains color.

**14. ABSTRACT**
The Blackjack project developed metrics and a test harness for evaluating compilers for scientific computing. The evaluation was performed by using representative applications and implementing relevant micro-benchmarks in order to test and analyze the productivity, correctness, and performance of multiple commercially available and freely available open source compiler systems.

**15. SUBJECT TERMS**
compilers, benchmarks, correctness, performance

| **16. SECURITY CLASSIFICATION OF:** | | | **17. LIMITATION OF ABSTRACT:** | **18. NUMBER OF PAGES** | **19a. NAME OF RESPONSIBLE PERSON** (Monitor) |
|---|---|---|---|---|---|
| **a. REPORT** | **b. ABSTRACT** | **c. THIS PAGE** | | | Alfred J. Scarpelli |
| Unclassified | Unclassified | Unclassified | SAR | 30 | **19b. TELEPHONE NUMBER** *(Include Area Code)* (937) 528-8898 |

**Standard Form 298 (Rev. 8-98)**
Prescribed by ANSI Std. Z39-18

## TABLE OF CONTENTS

## LIST OF FIGURES

# 1. SUMMARY

The Blackjack effort was part of the DARPA Architecture Aware Compiler Environment (AACE) program as a test and evaluation contract (T2) for the AACE compilers developers (T1). Due to a change in DARPA direction which eliminated the connection with the T1 teams, this effort changed focus to assess the capabilities of the existing commercial and free compilers. As the scope and functionality of currently available compilers is much different from the goals set out for the DARPA AACE compilers, Blackjack project objectives and methodology changed accordingly and only included a subset of original evaluation criteria and metrics. Specifically, a set of commercial and open source benchmarks, as well as applications, were chosen for evaluation of the existing compilers.

The problem under investigation is to assess the capabilities of commercially available and open source freely available compilers for compiling scientific software to produce correct and efficient executable versions for modern high performance computing (HPC) systems. The compilers tested were the Intel, and GNU compilers for C, C++, and Fortran. The conclusion was that while the different compilers have their strengths and weaknesses, none is clearly better than the others under all circumstances.

# 1 INTRODUCTION

The Blackjack project developed metrics and a test harness for evaluating compilers for scientific computing. The evaluation was performed by using representative applications and implementing relevant micro-benchmarks in order to test and analyze the productivity, correctness, and performance of multiple commercially available and freely available open source compiler systems. The main two goals were:

1. to test whether compiler technology can automatically select the appropriate optimizations based on a learned characterization of the target system; and
2. to ensure the compiler technology has a dynamic runtime environment that can dynamically improve the performance of a program during runtime and/or provide information that can be used by the compiler to optimize for future runs of the program.

The project consisted of two phases. Phase I developed a set of characterization benchmarks to determine platform characteristics such as cache sizes and memory latency and bandwidth.

The main focus of Phase II was the T2 team's evaluation of commercial and freely available production compilers. The evaluation of these compilers was based on measurements taken from the areas of performance, productivity, correctness and scalability. The Blackjack team defined evaluation metrics for each of the evaluation criteria in this list. The metrics were evaluated by using a test harness to run a set of test suites consisting of a range of benchmarks using the different compilers on two different test platforms.

# 2 METHODS, ASSUMPTIONS, AND PROCEDURES

## 2.1 System Characterization

BlackjackBench is a collection of portable micro-benchmarks that automate the characterization process and the statistical analysis techniques for interpreting the results. The BlackjackBench discovers the effective parameters of the hardware as experienced by the user application, rather than the often unattainable peak values. It aims at hardware characteristics that are observed by running executables from existing compilers and standard C codes. It characterizes the cache and Translation Lookaside Buffer (TLB) hierarchies, the cache sharing and Non Uniform Memory Access (NUMA) characteristics, the cost of arithmetic operations, the number of effective contexts (cores), the number of available registers, and the length of the OS scheduler time slot. We show how these features of modern multicore processors can be discovered programmatically. We also show how these features could potentially interfere with each other, which could result in incorrect interpretation of the results, and how established classification and statistical analysis techniques can reduce experimental noise and aid automatic interpretation of the results.

## 2.2 Evaluation Benchmarks and Metrics

The metrics we used for judging and scoring the production compilers are listed below by category:

- Correctness
    - Language conformance
    - Correct answers
- Robustness
    - Ease of installation
    - Response to errors
- Performance
    - Compile-time
        - Total time
        - Memory footprint
    - Runtime
        - Comparison to baseline benchmark execution time
        - Ability to apply performance-critical transformations

Phase II required three sets of test programs. The first was a set of correctness tests, to make sure the compilers correctly implement the language standards. For the base Fortran and C languages, we used language conformance tests. There are also tests for the OpenMP additions, and we can use other benchmark suites and community applications, to evaluate the robustness of the compilers.

The second set was to test the usability of the compiler. This included measuring compile time and resource usage and how well the compilers work with standard application build environments.

The third set, which overlapped the first two sets, tested for performance improvement. We selected several benchmarks and produced a baseline version of each for the target systems using the vendor or recommended compiler for that system. The baseline version was then used as the starting point for comparison.

More details concerning the benchmarks and metrics can be found in the Benchmarks and Metrics Specification Progress Report [2].

## 2.3 Evaluation Systems

We selected two evaluation systems, including Intel and AMD x86 multicore systems. The AMD system was `pluto.icl.utk.edu`, which is a Quad Processor AMD Magny Cours (48 cores) 6172. The Intel system was `zoot.icl.utk.edu`, which has four Intel Tigerton 2.4 GHz Processors (Quad Core) (16 total cores total).

## 2.4 Compiler Test Harness and Database

Blackjack has created a self-contained test harness that allows compilers to be tested both for correctness and performance, stores and logs the results within a central database, and allows access to results via a configurable Graphical User Interface (GUI). Using this test harness system allows compiler progress to be tracked and viewed using a flexible report-generation tool.

The creation of the test harness required numerous components, from basic makefiles, to Structured Query Language (SQL) queries, to web-page configuration. The management of the makefiles and the handling and coordination of all the items are performed with code written in Python. Using this approach, the overall system is portable and extensible.

At the most basic level, a generated test program needs to be executed on the chosen hardware running Unix-like Operating System (OS). To generate the test program, we need to follow several steps, and in general this process is makefile-driven. In particular, the makefile in conjunction with Python scripts comprises at least the following pieces:

1. **build** – this step incorporates the compiler, the given test, the options, and the desired end result (typically an object file).
2. **link** – the objects are linked together to produce an executable; options needed at this step include the linker to be used, a list of objects, linker flags, and library locations.
3. **run** – the executable is invoked on the desired platform. It may be sufficient to just name the executable, but there might also be arguments passed to the executable, or a command such as rsh might be needed if the executable is built on a host node (or a more involved process in a batch-queuing system might even be necessary).
4. **verify** – this step processes the output of the run step; depending on the type of test, execution time (and perhaps compile time) in addition to output results will be collected.
5. **record** – the data from the previous steps are stored within the database for later viewing and report generation.

The user interface for the test harness is web-based. The interface is the direct connection to the database, and may not exist on a remote-installed version of the test harness. When a user accesses the database, he/she may be uploading a results file from a remote location or accessing the database to view data and results. In this latter case there will be default reports generated on the web interface from which a user can examine or drill-down for more details. We also developed custom report generation capabilities that are useful for comparison purposes, as well as producing output suited to a particular evaluation task.

The main test harness command is runtest. An invocation of runtest specifies the test suite to be run along with optional arguments. The basic usage form is:

```
runtest -suite <N> [<options>] [-help]
```

Test suites are described in the next section.

More details about the test harness and database, including installation instructions, may be found in the documentation provided with the test harness release on the Blackjack website at `icl.eecs.utk.edu/blackjack`. As a result of deploying the test harness and database the user gains access to an automated system that can launch the compiler test suites from the convenience of a web page. Subsequently, the user can see the results (including execution time, command line output, etc.) on a web page by clicking appropriate link in the browser window. The results are archived in a database and the web interface is able to access this data based on the user settings and specific query details.

## 2.5 Test Suites

Tests and groups of tests fall into the following hierarchical scheme:

- At the top level a test grouping falls into a Test Suite Type, which describes the basic template for all tests or Suites of tests within the same type. Usually all tests or suites within the same Type share common characteristics of makefile structure, build sequence, execution method, and verification procedure (although a notable exception is the Application Suite Type, described in more detail below). Examples of Suite Types are C correctness tests, Fortran correctness tests, embedded tests, and SPEC tests.
- A Test Suite consists of one or more tests controlled by a given makefile. Information about the Suite will be stored in the database table suites. Examples of Suites are a particular set of correctness tests or a SPEC application instance.
- An individual test is simply an instance to be instantiated within a given Suite. It likely shows up as an item to be built and run within the makefile of its associated Suite.

It is important to note that performance applications generally consist of their own build procedures, run parameters, and verification methods. Thus it makes sense to group benchmarks that fall into the Application Suite Type together under the same Suite Type, despite the fact that there may be little similarity between their test cycle methods. This special grouping allows us to avoid creating Suite Types for each application benchmark.

# 3 RESULTS AND DISCUSSION

## 3.1 Hardware Characterization

One of the results of the project is BlackjackBench, a system characterization benchmark suite [1]. The contributions of this work are twofold:

1. A collection of portable micro-benchmarks that can probe the hardware and record its behavior while control variables, such as buffer size, are varied.
2. A statistical analysis methodology, implemented as a collection of scripts for result parsing, examines the output of the micro-benchmarks and produces the desired system characterization information, e.g. effective speeds and sizes.

BlackjackBench was specifically motivated by the effort to develop architecture aware compiler environments that automatically adapt to hardware that is unknown to the compiler writer and optimize application codes based on the discovery of the runtime environment. Often, important performance related decisions take into account effective values of hardware features, rather than their peak values. In this context, we consider an effective value to be the value of a hardware feature that would be experienced by a user level application written in C (or any other portable, high level, standards- compliant language) running on that hardware. This is in contrast with values that can be found in vendor documents, or through assembler benchmarks, or specialized instructions and system-calls.

BlackjackBench goes beyond the state of the art in system benchmarking by characterizing features of modern multicore systems, taking into account contemporary, complex, hardware characteristics such as modern sophisticated cache prefetchers, the interaction between the cache and TLB hierarchies, etc. Furthermore, BlackjackBench combines established classification and statistical analysis techniques with heuristics tailored to specific benchmarks, to reduce experimental noise and aid automatic interpretation of the results. As a consequence, BlackjackBench does not merely output large sets of data that require human intervention and comprehension; it shows information about the hardware characteristics of the tested platform. Moreover, BlackjackBench does not rely on assembler code, specialized kernel modules and libraries, or non-portable system calls. Therefore, it is a portable system characterization tool.

### 3.1.1 Cache Hierarchy

Improved cache utilization is one of the most performance critical optimizations in modern computer hardware. Processor speed has been increasing faster than memory speed for several decades, making it increasingly harder for main memory to feed all processing elements with data quickly enough. To bridge the gap, fast, albeit small, cache memory has become necessary for fast program execution. In recent years, the pressure on main memory has increased further, as the number of processing elements per socket has been going up. As a result, most modern processor designs incorporate complex, multi-level cache hierarchies that include both shared and non-shared cache levels between the processing elements. Selected results from the cache hierarchy benchmarks are shown in Figures 1, 2, and 3. More details can be found in a published paper about the benchmark [1].

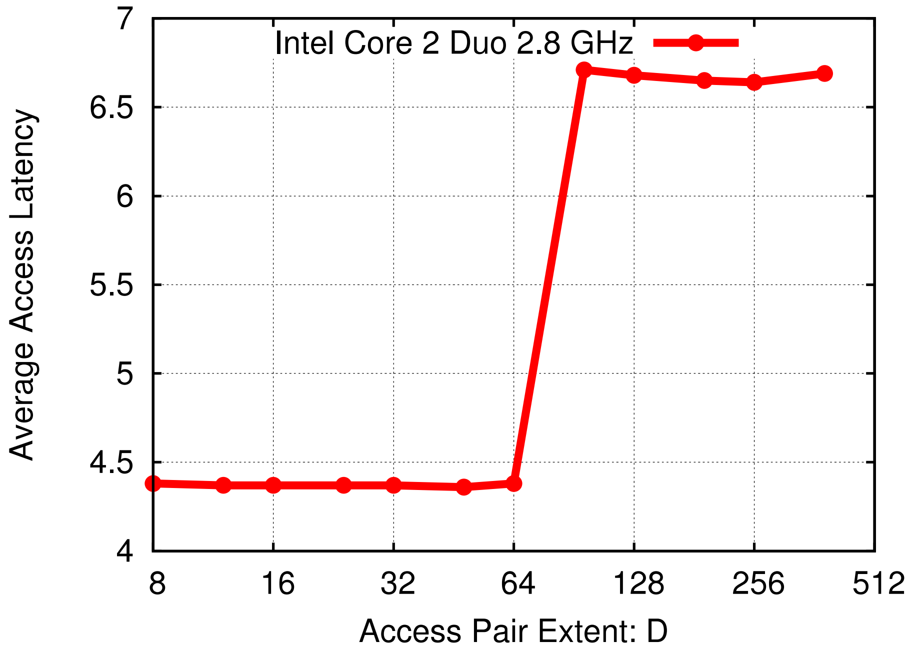


**Figure 1: Cache Line Size Characterization on Intel Core 2 Duo.**

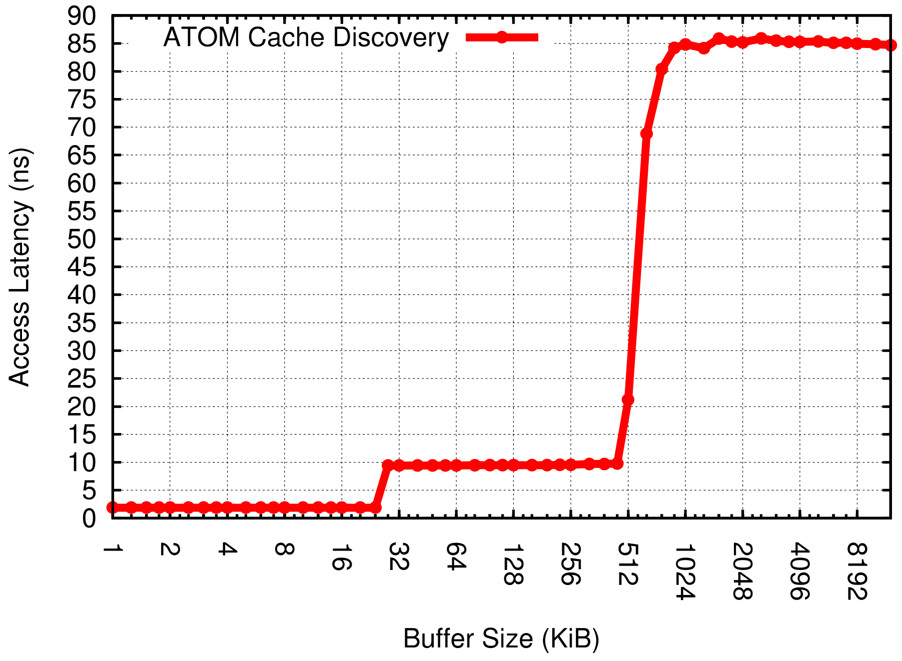


**Figure 2: Cache Count, Size and Latency Characterization on Intel Atom.**

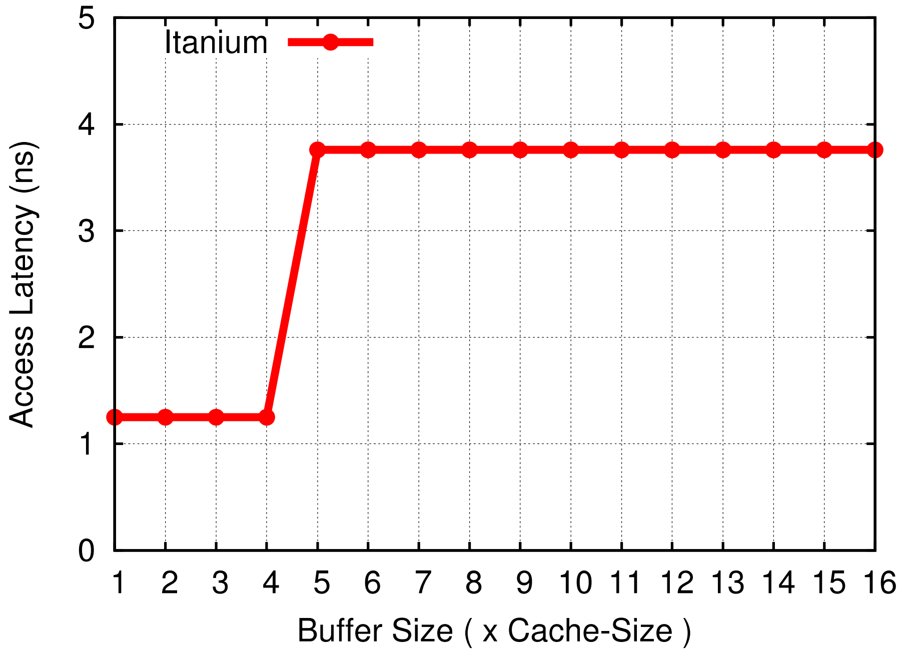


**Figure 3: Cache Associativity Characterization on Itanium II.**

### 3.1.2 Asymmetries in the Memory Hierarchy

With the move to multi-core processors, we witnessed the quasi- general introduction of shared cache levels to the memory hierarchy. A shared cache design provides larger cache capacity by eliminating data replication for multi-threaded applications. The entire cache may be used by a single active core for single-threaded workloads. More importantly, a shared cache design eliminates on- chip cache coherence at that cache level. In addition, it resolves coherence of the private lower level caches internally within the chip and thus reduces external coherence traffic. One downside of shared caches is a larger hit latency, which may cause increased cache contention and unpredictable conflicts. Shared caches are not an entirely new design feature. Before level two caches were integrated onto the chip, some Symmetric Multiprocessor (SMP) architectures were using external shared L2 caches to increase capacity for single threaded workloads, and to reduce communication costs between processors.

Figure 4 shows aggregated results for an Intel Gainestown system with two sockets and Hyper-Threading disabled. The X axis represents the memory block size, and the y axis represents the bandwidth observed by one of the threads. The bandwidth is computed as number_updated_lines * cache_line_size / time, where number_updated_lines is the number of cache lines updated by the first thread. Since the two threads update an equal number of lines, the values shown in the figure represent only half of the actual two-way bandwidth.

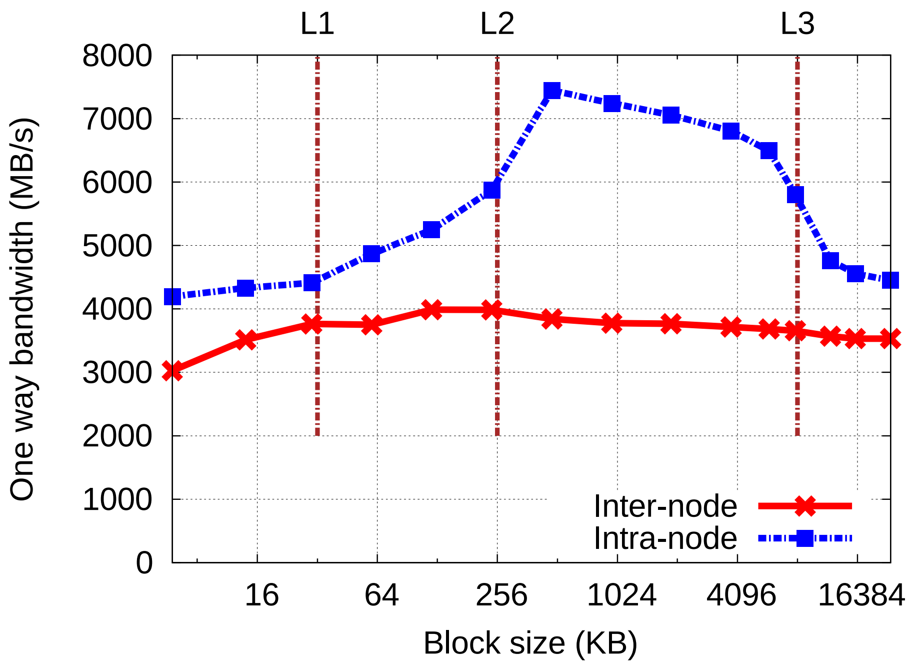


**Figure 4: One-way, inter-core communication bandwidth for different memory block sizes and core placements on a dual-socket Intel Gainestown system.**

### 3.1.3 TLB Hierarchy

TLB hierarchy is an important part of the memory system that bears some resemblance to the cache hierarchy. However, TLB is sufficiently different to warrant its own characterization methodology. Accordingly, we will focus on the description of our TLB benchmarking techniques rather than present differences and similarities with the cache benchmarks.

The crucial feature that any TLB benchmark should posses is the ability to alleviate cache effects on the measurements. Both conflict and capacity misses coming from data caches should either be avoided at runtime or filtered out during the analysis of the results. We chose the latter, as it has the added benefit of capturing the rare events when the TLB and the data cache are inherently interconnected, such as when TLB fits the same number of pages as there are data cache lines.

To determine the page size, our benchmark maximizes the penalty coming from the TLB misses. We do it by traversing a large array multiple times with a given stride. The array is large enough to exceed the span of any TLB level – this guarantees a high miss rate if the stride is larger or equal to the page size. If the stride is less than the page size, some of the accesses to the array will be contained in the same page, and thus, will decrease the number of misses and the overall benchmark execution time. The false positives stemming from interference of data cache misses are eliminated by the high cost of a TLB miss in the last level of TLB. Handling these misses requires the traversal of the OS page table stored in main memory – the combined latency exceeds the cost of a miss for any level of cache. Typical timing curves for this benchmark are shown in Figure 5. The figure shows results from three very different processors: ARM OMAP3, Intel Itanium 2, and Intel Nehalem EP. The graph line for each system has the same shape; for strides smaller than the page size the line raises as the number of misses increases because fewer memory accesses hit the same page. And for strides that exceed the page size, the graph line is flat because each array access touches a different page so the per-access overhead remains the same. The page size is determined as the inflection point in the graph line. For our example, the page size is 4 KiB on both ARM and Nehalem, and 16 KiB on Itanium.

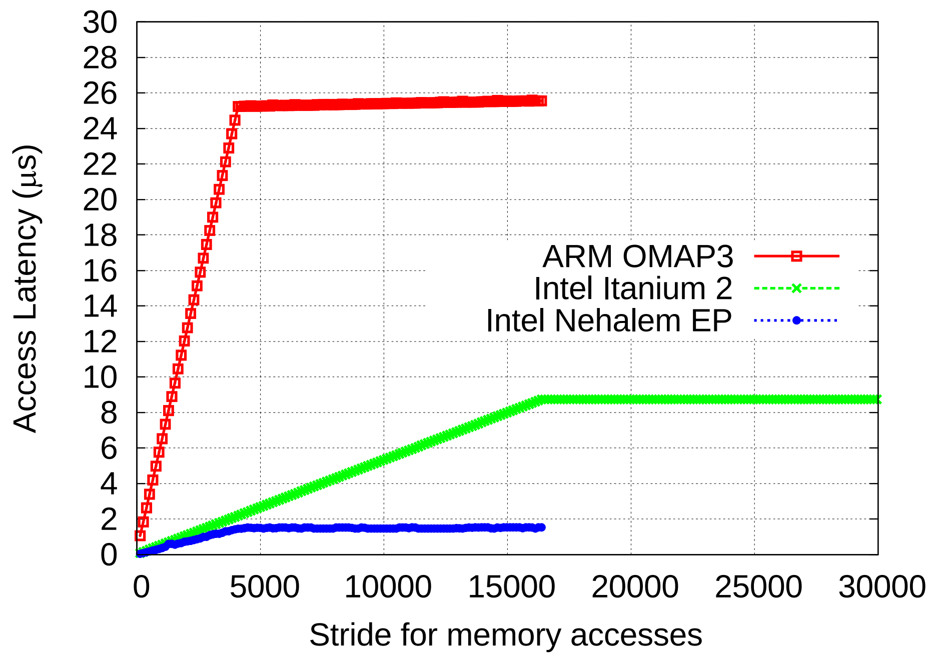


**Figure 5: Graph of timing results that reveal the TLB page size.**

Once the actual TLB page is known, it is possible to proceed with discovering the sizes of the levels of the TLB hierarchy. Probably the most important task is to minimize the impact of the data cache. The common and portable technique is to perform repeated accesses to a large memory buffer at strides equal to the TLB page size. This technique is prone to creating as many false positives as there are data cache levels, and a slight modification to this technique is required. On each visited TLB page, our benchmark chooses a different cache line to access, thus, maximizing the use of any level of data cache. As a side note, choosing a random cache line within a page utilizes only half of the data cache on average. Figure 6 shows the timing graphs on a variety of platforms. Both Level 1 and Level 2 TLBs are identified accurately.

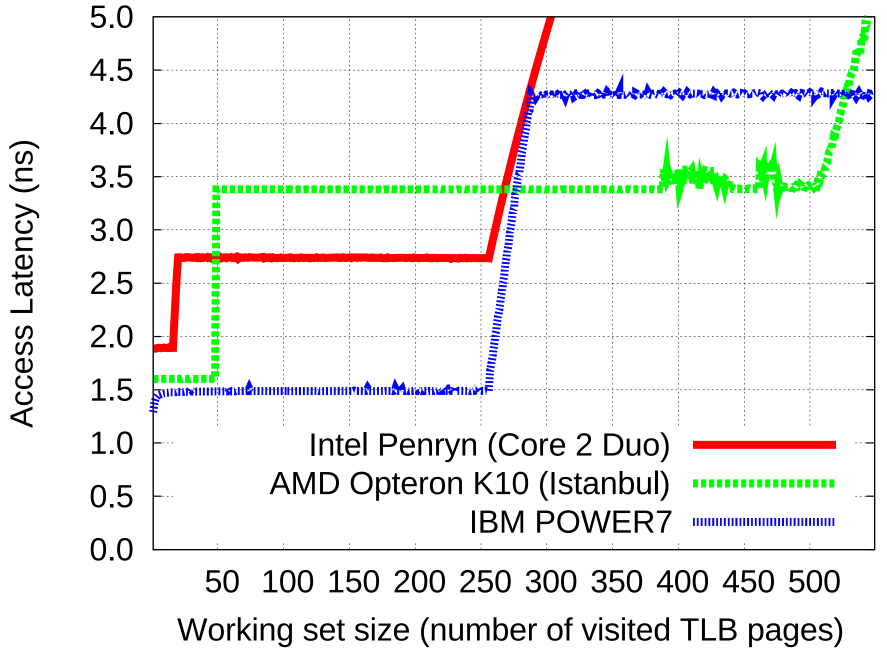


**Figure 6: Graph of results that reveal the number of levels of TLB and the number of entries in each level.**

### 3.1.4 Execution Contexts

Modern systems commonly have multiple cores per socket and multiple sockets per node. To avoid confusion due to the overloading of the terms core, CPU, node, etc., by hardware vendors, we use the term “Execution Context” to refer to the minimum hardware necessary to effect the execution of a compute thread. Several modern architectures implementing virtual hardware threads exhibit selective preference over different resources. For example, a processor could have private integer units for each virtual hardware thread, but only a shared floating-point unit for all hardware threads residing on a physical core. The Blackjack benchmarks attempt to discover the maximum number of:

1. Floating Point Execution Contexts
2. Integer Execution Contexts
3. Memory Intensive Execution Contexts

A sample output of this benchmark on an IBM POWER7 processor can be seen in Figure 7.

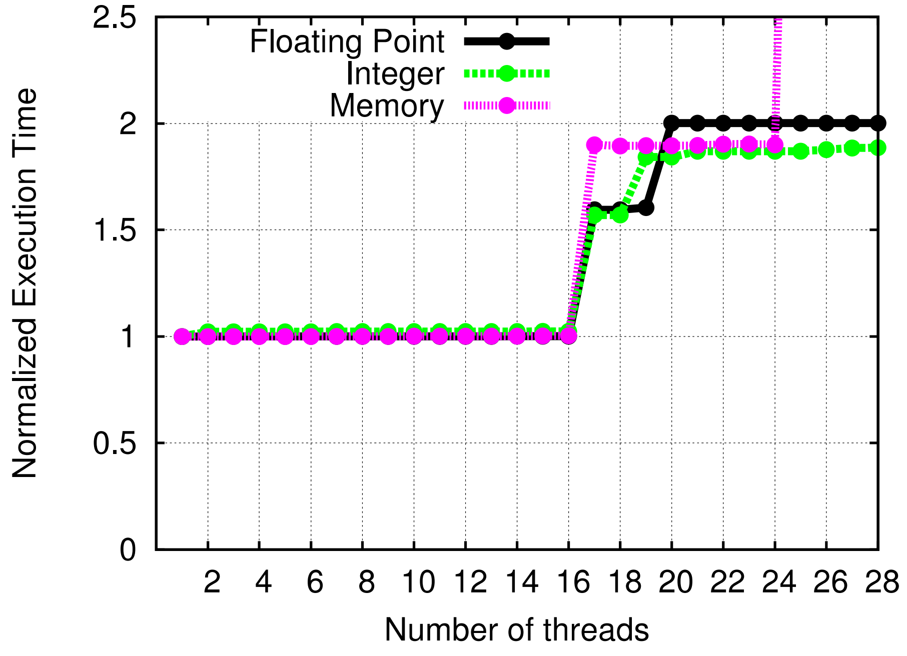


**Figure 7: Execution Contexts Characterization on IBM POWER7.**

### 3.1.5 Statistical Analysis

The output of the micro-benchmarks discussed above is typically a performance curve, or rather, a large number of points representing the performance of the code for different values of the control variable. Since our motivation for developing BlackjackBench was to inform compilers and auto-tuners about the characteristics of a given hardware platform, we have developed analyses that can process the performance curves and output values that correspond to actual hardware characteristics. More details and examples may be found in a published paper about the benchmark [1].

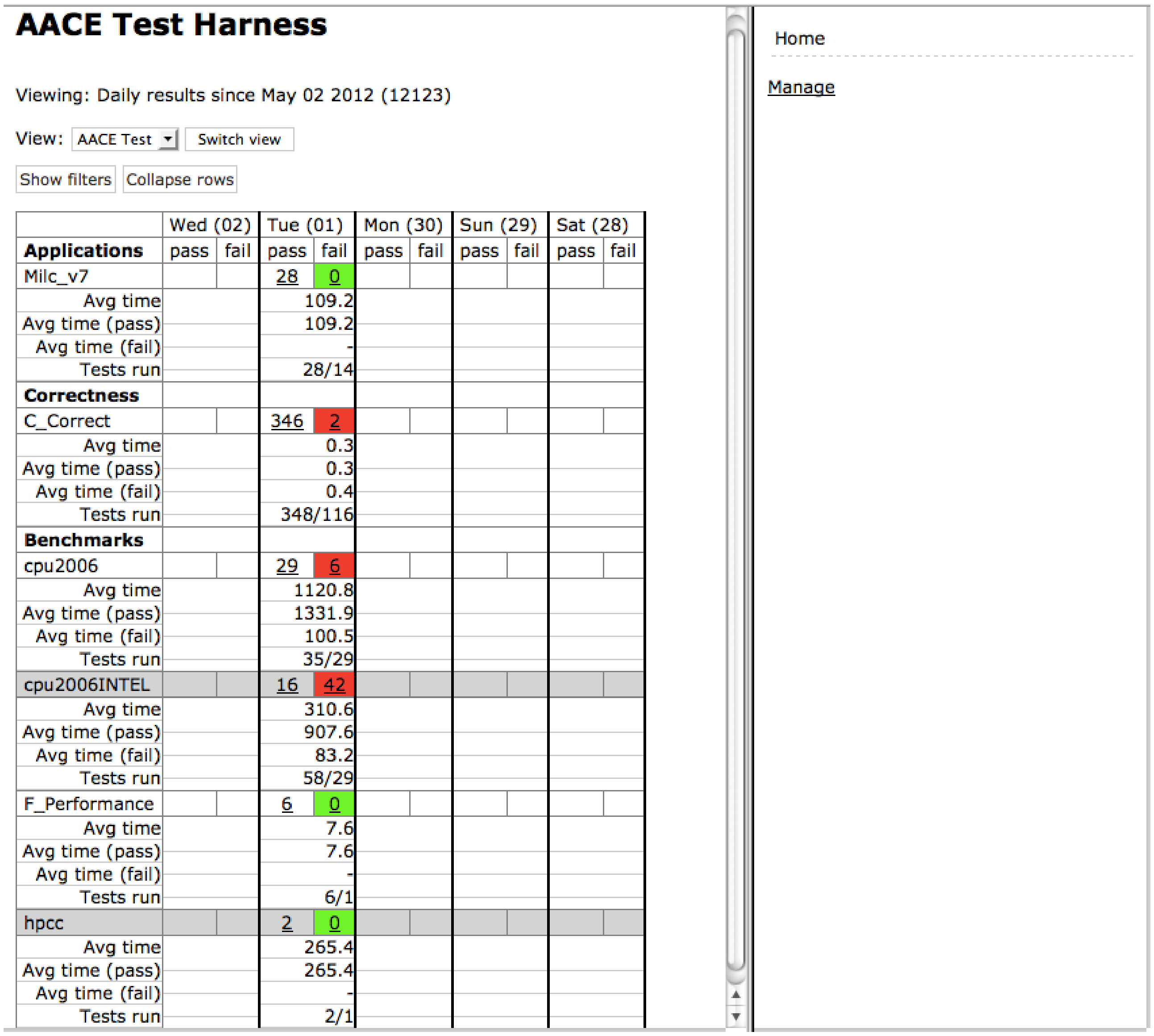
## 3.2 Compiler Evaluation

### 3.2.1 Test Harness Results

As can be seen in Figure 8, we used the Blackjack Test Harness with several benchmark code bases. In particular, we used:

- C_Correct, a suite of correctness tests for C syntax.
- MILC ver. 7, a set of codes written in C for doing simulations of four dimensional SU(3) lattice gauge theory.
- SPEC CPU 2006, a popular performance test for processors by the Standard Performance Evaluation Corporation.
- SPEC CPU 2006 Intel, a variant of SPEC CPU 2006 for Intel processors.

- HPCC, the High Performance Computing Challenge Benchmark, a DARPA funded benchmark suite that provides a comprehensive way to assess the overall performance of HPC machines.
- F_Performance, a performance test for a Fortran code.



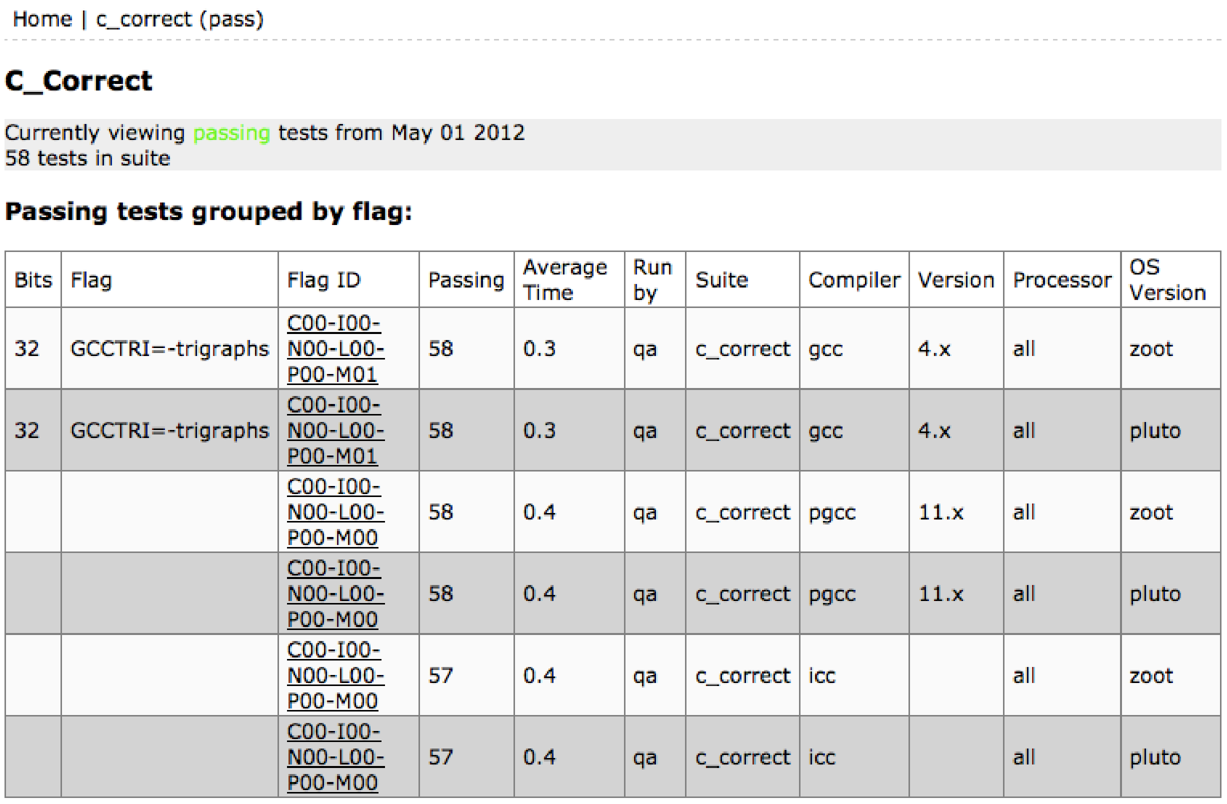
**AACE Test Harness**

Viewing: Daily results since May 02 2012 (12123)

|  | Wed (02) |  | Tue (01) |  | Mon (30) |  | Sun (29) |  | Sat (28) |  |
|---|---|---|---|---|---|---|---|---|---|---|
| **Applications** | pass | fail | pass | fail | pass | fail | pass | fail | pass | fail |
| Milc_v7 |  |  | 28 | 0 |  |  |  |  |  |  |
| Avg time |  |  | 109.2 |  |  |  |  |  |  |  |
| Avg time (pass) |  |  | 109.2 |  |  |  |  |  |  |  |
| Avg time (fail) |  |  | - |  |  |  |  |  |  |  |
| Tests run |  |  | 28/14 |  |  |  |  |  |  |  |
| **Correctness** |  |  |  |  |  |  |  |  |  |  |
| C_Correct |  |  | 346 | 2 |  |  |  |  |  |  |
| Avg time |  |  | 0.3 |  |  |  |  |  |  |  |
| Avg time (pass) |  |  | 0.3 |  |  |  |  |  |  |  |
| Avg time (fail) |  |  | 0.4 |  |  |  |  |  |  |  |
| Tests run |  |  | 348/116 |  |  |  |  |  |  |  |
| **Benchmarks** |  |  |  |  |  |  |  |  |  |  |
| cpu2006 |  |  | 29 | 6 |  |  |  |  |  |  |
| Avg time |  |  | 1120.8 |  |  |  |  |  |  |  |
| Avg time (pass) |  |  | 1331.9 |  |  |  |  |  |  |  |
| Avg time (fail) |  |  | 100.5 |  |  |  |  |  |  |  |
| Tests run |  |  | 35/29 |  |  |  |  |  |  |  |
| cpu2006INTEL |  |  | 16 | 42 |  |  |  |  |  |  |
| Avg time |  |  | 310.6 |  |  |  |  |  |  |  |
| Avg time (pass) |  |  | 907.6 |  |  |  |  |  |  |  |
| Avg time (fail) |  |  | 83.2 |  |  |  |  |  |  |  |
| Tests run |  |  | 58/29 |  |  |  |  |  |  |  |
| F_Performance |  |  | 6 | 0 |  |  |  |  |  |  |
| Avg time |  |  | 7.6 |  |  |  |  |  |  |  |
| Avg time (pass) |  |  | 7.6 |  |  |  |  |  |  |  |
| Avg time (fail) |  |  | - |  |  |  |  |  |  |  |
| Tests run |  |  | 6/1 |  |  |  |  |  |  |  |
| hpcc |  |  | 2 | 0 |  |  |  |  |  |  |
| Avg time |  |  | 265.4 |  |  |  |  |  |  |  |
| Avg time (pass) |  |  | 265.4 |  |  |  |  |  |  |  |
| Avg time (fail) |  |  | - |  |  |  |  |  |  |  |
| Tests run |  |  | 2/1 |  |  |  |  |  |  |  |

**Figure 8: Code bases used with the Compiler Test Harness.**

In Figure 9, we can see that the harness can test several compilers, against each benchmark, to evaluate their quality.


Home | c_correct (pass)

**C_Correct**

Currently viewing passing tests from May 01 2012
58 tests in suite

**Passing tests grouped by flag:**

| Bits | Flag | Flag ID | Passing | Average Time | Run by | Suite | Compiler | Version | Processor | OS Version |
|---|---|---|---|---|---|---|---|---|---|---|
| 32 | GCCTRI=-trigraphs | C00-I00-N00-L00-P00-M01 | 58 | 0.3 | qa | c_correct | gcc | 4.x | all | zoot |
| 32 | GCCTRI=-trigraphs | C00-I00-N00-L00-P00-M01 | 58 | 0.3 | qa | c_correct | gcc | 4.x | all | pluto |
|  |  | C00-I00-N00-L00-P00-M00 | 58 | 0.4 | qa | c_correct | pgcc | 11.x | all | zoot |
|  |  | C00-I00-N00-L00-P00-M00 | 58 | 0.4 | qa | c_correct | pgcc | 11.x | all | pluto |
|  |  | C00-I00-N00-L00-P00-M00 | 57 | 0.4 | qa | c_correct | icc |  | all | zoot |
|  |  | C00-I00-N00-L00-P00-M00 | 57 | 0.4 | qa | c_correct | icc |  | all | pluto |

**Figure 9: Correctness test of multiple compilers.**

Since the AACE program terminated before producing the experimental compilers for which we had developed the test harness, we demonstrate the functionality of the harness on the open source compiler `gcc` and the commercial compilers `icc` from Intel, respectively. Also, in the figure we can see that results from different platforms (`zoot`, `pluto`) can be stored into the database of the harness and presented to the evaluator in a single page.

### 3.2.2 Interpretation of Results

#### C_correctness

The C_correctness test is meant to ascertain the compiler's ability to parse a large set of language constructs. It is not, by far, the most exhaustive test. Only the Intel C compiler properly identified an error that tested what happens when there is a return statement missing in function returning "`void`". This is the relevant error message from the Intel compiler `icc`:

```
s3.c(13): warning #1011: missing return statement at end of non-void function
"test"
```

On one hand, the newest C standard deems such constructs as errors and the compiler has every right to complain about it. On the other hand, such statements are a prevailing practice and are deeply entrenched in fairly large code bases. Following this observation, the `gcc` compiler does not consider this construct to be worth issuing a warning unless instructed specifically by the user with an appropriate command line option.

Aside from the above ambiguity, the remaining 57 of 58 tests pass without problems on both tested systems, with both compilers.

## MILC ver. 7

We included 28 tests in our harness that use many features that could stress both the harness and the compiler. MILC is a physics application with a large code base. Due to its size, it could create a problem for the harness during both build and running stages. The number of files that are built for each test could be as high as 50 and the total number of lines of code is counted in thousands. In addition, MILC requires setup to properly account for optimization options and the available build infrastructure. We used the tests from MILC to test the behavior of the harness under various failing conditions. And the conditions included:

- Lack of proper Makefile in one of the MILC source code directories
- Exceeding the time available for the build or running stages

In all cases, the harness behaved as expected by recording the incident in the database and making the failing output available for inspection. This is in line with the design objective of having enough information available in the database to properly diagnose and even correct the error.

## SPEC CPU 2006

SPEC CPU 2006 is a benchmark that tests system performance by running a comprehensive suite of applications and providing appropriately chosen weighted averages to aide in direct comparisons of computers. The following components were used in tests:

- **astar** (C++) path finding algorithms.
- **bwaves** (Fortran) simulation of blast waves in 3D transonic transient laminar viscous flow.
- **bzip2** (C) in-memory compression using block-sorting lossless algorithm for Joint Photographic Experts Group (JPEG) images, a code binary, tar file of source code, and an Hyper Text Markup Language (HTML) file.
- **cactusADM** (Fortran 90, C) Einstein evolution equation solver based on Cactus open source code and BenchADM (for Arnowitt, Desser, and Misner formalism) computational kernels.
- **calculix** (Fortran 90, C) finite element code for linear and non-linear 3D structural applications mostly used in structural mechanics.
- **dealII** (C++) finite element code with error estimators and adaptive meshes that uses advanced C++ features.
- **gamess** (Fortran) quantum mechanical code for self-consistent field computations.
- **gcc** (C) source code for `gcc` version 3.2.

- **GemsFDTD** (Fortran 90) solver for 3D Maxwell equations using finite-difference time-domain method.
- **gobmk** (C) analyzer for Go game positions.
- **gromacs** (Fortran and C) molecular dynamics code that performs simulation of the Newtonian equations of motion for systems of with hundreds to millions of particles.
- **h264ref** (C) video compression algorithm.
- **hmmer** (C) genome sequence search based on Profile Hidden Markov Models.
- **lbm** (C) implementation of Lattice Boltzman Method for simulation of incompressible fluids in 3D.
- **leslie3d** (Fortran 90) research-level Computational Fluid Dynamics code for Large-Eddy Simulations with Linear-Eddy Model in 3D.
- **libquantum** (C) simulator of a quantum computer.
- **mcf** (C) a solver for single-depot vehicle scheduling problem in public mass transportation.

Each of these components is commonly used as a separate application and usually requires input data. SPEC CPU 2006 provides these input data sets for each component application.

The results for SPEC CPU 2006 consist of 5 columns. Each column is defined in length by SPEC on their website: http://www.spec.org/auto/cpu2006/Docs/result-fields.html. We give here a sufficient description of each column to make this document self-contained:

1. **ID**: each test included in SPEC CPU 2006 suite is given a unique identification number that can easily be used in the suite of tools provided by SPEC.
2. **NAME**: each test included in SPEC CPU 2006 suite is given a unique name is primarily meant for readability of the results.
3. **Base Time** (seconds): each test was run on a reference platform (Sun UltraSparc II system at 296 MHz) and this time is printed for reference.
4. **Execution Time** (seconds): each test is run 3 times and the median time is recorded as a primary performance metric. Obviously, the smaller the execution time is the faster the tested system.
5. **Ratio**: to provide unit-less measure of the tested system performance the ratio of base time to execution time is printed.

In our tests, many component applications exceeded 1000 seconds of running time, sometimes even approaching the 2000-second mark. In addition they produce a fairly large output at the console screen. Neither of these caused issues for the harness, and the results were properly recorded in the database.

The problems recorded by the harness occurred in the astar, bwaves, and dealII component applications. The errors may be attributed to general classes: linking and C++ instantiation. The former has to do with linking against language specific libraries such as math library for C and Gfortran intrinsics for Fortran. The latter is related to instantiation of specific member function in C++ classes used by the code. As before, the harness properly detects the occurrence of failure and preserves the command line output in the database, which is then accessible through the web interface of the harness.

**SPEC CPU 2006 Intel**

This streamlined version of the SPEC CPU 2006 benchmark suite is targeted specifically for Intel software and hardware. Naturally, a lot of errors recorded by the harness have to do with proper configuration of the tests and accounting for proper system settings. This can be observed by consulting the specific sections of the harness' GUI that relate to this particular test.

### 3.2.3 Discussion of SPEC CPU 2006 Results

We have used the SPEC CPU 2006 suite of tests to perform evaluation of both the harness and the compilers themselves. Test suites coming from SPEC are comprehensive in both their depth and breadth, and as a result, produce a quite substantial number of performance metrics to allow for informed comparisons between tested systems. At the same time, however, the members of SPEC recognize the value of conciseness and offer various methods of aggregating the reported results with careful attention to preserve balanced insight into the systems' performance. When quoting the SPEC CPU 2006 results in this report, we chose to show the values produced from individual runs of each test. Each such run was executed separately by the Blackjack harness, and was recorded successfully in the associated database. The failed attempts were used only to test the robustness of the harness and are not used for the compiler evaluation shown in this section.

**Results from `gcc`.** To test the `gcc` compiler, we used version 4.1 with 64-bit support, which allowed the compiler to use a larger register file than is normally available in the 32-bit mode. The performance results reported by the various components of the SPEC CPU 2006 suite are shown below:

```
 ID.NAME      Base Time   Exec Time    Ratio
----------------------------------------------------------------------------
401.bzip2        9650        914       10.6 *
403.gcc          8050        701       11.5 *
410.bwaves      13590       2576       5.28 *
416.gamess      19580       1525       12.8 *
435.gromacs      7140        923       7.74 *
436.cactusADM   11950       1957       6.11 *
437.leslie3d     9400        911       10.3 *
445.gobmk       10490        805       13.0 *
454.calculix     8250       3046       2.71 *
456.hmmer        9330       1098       8.50 *
459.GemsFDTD    10610       1059       10.0 *
462.libquantum  20720       1060       19.5 *
464.h264ref     22130       1302       17.0 *
470.lbm         13740        849       16.2 *
```

The noteworthy flags included "`-DSPEC_CPU_LP64`" and "`-O2`". The "`-DSPEC_CPU_LP64`" flag enables 64-bit mode for the source code, and assumes that both "`long`" integral values and pointers are 64-bit entities which was the case for our tested machines. The optimization option chosen for the run was "`-O2`".

**Results for Intel compiler.** We tested a full commercial version of the Intel compiler. The exact version number was 2011.6.233. The flag for enabling 64-bit processing was "-m64".   The set of optimization options was the commonly used shorthand flag “-fast”. The performance results reported by the Intel compiler are as follows:

```
 ID.NAME      Base Time   Exec Time    Ratio
--------------------------------------------------------------------------------
416.gamess      19580       1335       14.7 *
434.zeusmp       9100        592       15.4 *
435.gromacs      7140        590       12.1 *
436.cactusADM   11950        748       16.0 *
437.leslie3d     9400        666       14.1 *
454.calculix     8250        491       16.8 *
459.GemsFDTD    10610        831       12.8 *
465.tonto        9840        588       16.7 *
```

**Discussion.** The most immediate observation is the fact that the Intel compiler always delivers faster performance than the GNU compiler. This is the case despite the fact that the tests were performed on an AMD processor. The advantage that the Intel compiler holds over the GNU compiler may be as low as 20% (see 416.gamess results: 1335 seconds versus 1525 seconds) but it may be as high as many-fold more (see 454.calculix: 491 seconds versus 3046 seconds). We attribute such stark differences to a better instruction generation at the very backend of the compilation stages. We would definitely expect this difference to be even larger for an Intel processor.

On the negative side, we observe that the `gcc` compiler is capable of successfully completing many more tests (14 in total) than the Intel compiler (8 in total). Lack of successful completion can either mean that the compilation failed (e.g., due to complicated use of advanced C++ syntax) or the result does not verify by using the test’s internal consistency checks. The difference in completion rates may be attributed to more aggressive optimization levels enabled by the “`-fast`” option for the Intel compiler. However, we chose this option as the recommended optimization setting indicated by the Intel compiler documentation. This flag is in fact very likely to be selected by most end users.

## HPC Challenge

HPC Challenge (HPCC) is a benchmark that tests floating point performance of a distributed memory machine under different memory access patterns. HPCC depends on Message Passing Interface (MPI) and requires a special command to run the test. The Blackjack Harness may be easily configured for such a case, which is why there was a lack of failures recorded for HPCC in our web harness.

## F_Performance

F_Performance includes only a single performance test that measures running time of a generic matrix-matrix multiplication routine. The harness records related to F_Performance indicated successful test due to its small code size and few features that are exercised from the Fortran language.

# 4 CONCLUSION

## 4.1 Hardware Characterization

We have presented the BlackjackBench system characterization suite. This suite of micro-benchmarks goes beyond the state-of-the-art in benchmarking by:

1. Offering micro-benchmarks that can exercise a wider set of hardware features than most existing benchmark suites do.
2. Emphasizing portability by avoiding low-level primitives, specialized software tools and libraries, or non-portable OS calls.
3. Providing comprehensive statistical analyses as part of the characterization suite, capable of distilling the micro-benchmarks' results into useful values that describe the hardware.
4. Emphasizing the detection of hardware features through variations in performance. As a result, BlackjackBench detects the effective values of hardware characteristics, which is what a user level application experiences when running on the hardware, instead of often unattainable peak values.

We described how the micro-benchmarks operate and their fundamental assumptions. We explained the analysis techniques for extracting useful information from the results and demonstrated through several examples, drawn from a variety of hardware platforms and operating systems, that our assumptions are valid and our benchmarks portable.

## 4.2 Compiler Evaluation

We have used the Blackjack compiler test harness with a collection of correctness and performance benchmarks to evaluate the GNU and Intel compilers. The results can be viewed from a Web browser. The results show that while the Intel compiler in general produces better performance, the GNU compiler is more robust.

## 5 REFERENCES

1. Anthony Danalis, Piotr Luszczek, Gabriel Marin, Jeffrey S. Vetter, and Jack Dongarra, BlackjackBench: Portable Hardware Characterization, Special Issue of ACM Performance Evaluation Review, 40(2), 2012.
2. Anthony Danalis, Piotr Luszczek, and Shirley Moore, Benchmarks and Metrics Specification Progress Report, DARPA AACE Program Phase II, 2011.

## List of Acronyms, Abbreviations, and Symbols

| **Acronym** | **Description** |
|---|---|
| AACE | Architecture Aware Compiler Environment |
| ACM | Association for Computing Machinery |
| AMD | Advanced Micro Devices |
| ARM | Acorn RISC Machine |
| CPU | Central Processing Unit |
| GCC | GNU Compiler Collection |
| GNU | Gnu is Not Unix |
| GUI | Graphical User Interface |
| HPC | High Performance Computing |
| HPCC | HPC Challenge |
| IBM | International Business Machines |
| ICC | Intel C Compiler |
| JPEG | Joint Photographic Experts Group |
| KiB | kibibyte, 1024 bytes |
| MPI | Message Passing Interface |
| NUMA | Non Uniform Memory Access |
| OMAP | Open Multimedia Applications Platform |
| OS | Operating System |
| OpenMP | Open Multi Processing |
| RISC | Reduced Instruction Set Computer |
| SMP | Symmetric Multiprocessor |
| SPEC | Standard Performance Evaluation Corporation |
| SQL | Structured Query Language |
| TLB | Translation Lookaside Buffer |
| 3D | three dimensional |